Highly Efficient Compensation-based Parallelism for Wavefront Loops on GPUs

Kaixi Hou, Hao Wang, Wu-chun Feng
Dept. Computer Science, Virginia Tech
Blacksburg, VA, USA
{*kaixihou, hwang121, wfeng*}*@vt.edu*

Jeffrey S. Vetter, Seyong Lee
Oak Ridge National Lab
Oak Ridge, TN, USA
vetter@computer.org, lees2@ornl.gov

***Abstract*—Wavefront loops are widely used in many scientific applications, e.g., partial differential equation (PDE) solvers and sequence alignment tools. However, due to the data dependencies in wavefront loops, it is challenging to fully utilize the abundant compute units of GPUs and to reuse data through their memory hierarchy. Existing solutions can only optimize for these factors to a limited extent. For example, tiling-based methods optimize memory access but may result in load imbalance; while compensation-based methods, which change the original order of computation to expose more parallelism and then compensate for it, suffer from both global synchronization overhead and limited generality.**

In this paper, we first prove under which circumstances that breaking data dependencies and properly changing the sequence of computation operators in our compensation-based method does *not* affect the correctness of results. Based on this analysis, we design a highly efficient compensation-based parallelism on GPUs. Our method provides weighted scan-based GPU kernels to optimize the computation and combines with the tiling method to optimize memory access and synchronization. The performance results on the NVIDIA K80 and P100 GPU platforms demonstrate that our method can achieve significant improvements for four types of real-world application kernels over the state-of-the-art research.

I. Introduction

Modern accelerators, e.g., GPUs, feature wide vector-like compute units and a complex memory hierarchy. If parallel applications can be organized to follow the SIMD processing paradigm, coalesced memory access patterns, and data reuse at different levels of the memory hierarchy, GPUs can often deliver superior performance over CPUs. However, *wavefront loops*, which can be found in many scientific applications, including partial differential equation (PDE) solvers and sequence alignment tools, are exceptions. Because their computations (including the association operator and the distribution operator, discussed in Sec. II-A) update each entry of a two-dimensional (2-D) matrix based on the already-updated values from its upper, left, and (optional) diagonal neighbors, this strong data dependency hinders the optimizing of computation and memory access on GPUs at the same time. That is, if data is stored in a row- or column-major order, the data can be processed in parallel but from non-contiguous memory addresses. In other words, data dependencies prevent consecutively stored data to be processed in parallel. Alternatively, if the data is stored in an anti-diagonal-major order, parallel computation can naturally follow the data dependency, but the exposed parallelism may result in severe load imbalance and underutilization of the compute units.

Fig. 1 provides an overview of existing approaches to optimize waverfront loops on GPUs. In ①, the parallelism on anti-diagonal data is directly exposed by applying loop transformation techniques, e.g., loop skewing and loop interchange [1, 2]. These methods may lead to severe performance penalties due to non-contiguous memory access and load imbalance. In ②, the tiling-based methods [3, 4, 5] block the reusable data in local memories, e.g., caches, to reduce the overhead of uncoalesced memory access. However, as their computation still strictly follows the anti-diagonal order, the load imbalance occurs at the beginning and ending anti-diagonals, causing the underutilization of computing resources. Although the recent tiling-based method called PeerWave [5] can mitigate the problem, it incurs extra overhead to transform the data layout. In contrast, instead of mainly optimizing memory accesses on GPUs, the recent studies [6, 7] in ③ focus on accelerating computations and resolving the load imbalance. The core idea is to ignore the data dependency along a row at first, compute data entries in a row in parallel, and finally correct the intermediate results. We call this method as a **compensation-based** parallelism for wavefront loops. However, this method requires expensive global synchronizations within and between processing each row, leading to frequent global synchronizations and the loss of data reuse. More importantly, because this method does not follow the original data dependency and has changed the sequence of computation operators on data entries, the domain knowledge from developers is required for the correctness of the final results, which makes it a privilege for experienced users only.

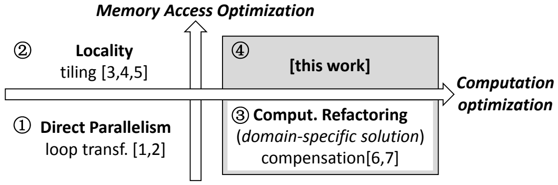


Figure 1: Parallelization landscape for wavefront loops.

In this paper, we first investigate under which circumstances, the compensation-based method that breaks through the data dependency and changes the sequence of computation operators properly can be used to optimize wavefront loops. We prove that if the accumulation operator is associative and commutative and the distribution operator is either distributive over or same with the accumulation operator, changing the sequence of operators properly doesn't affect the correctness of results. We also analyze that several popular algorithms, including a successive over-relaxation (SOR) solver [8], Smith-Waterman (SW) algorithm [9], summed-area table (SAT) computation [10], and integral histogram (IHist) algorithm [11], satisfy such requirements. Due to its generality, we design a highly efficient compensation-based solution for wavefront loops on GPUs: we propose a weighted scan-based method to accelerate the computation and combine it with the tiling method to optimize memory access and reduce global synchronization overhead.

In the evaluation, we first compare the performance of the weighted scan-based GPU kernels with those based on widely-used libraries, i.e., Thrust [12] and ModernGPU [13], and our kernels can deliver an average of 3.5x and 4.7x speedups on NVIDIA K80 and P100 GPUs, respectively. We also use our methods to optimize SOR, SW, SAT, and IHist application kernels, yielding up to 22.1x (43.3x) speedups on K80 (P100) over state-of-the-art optimizations [5]. Even for their best scenarios, we can still obtain an average of 1.1x (1.8x) improvements on K80 (P100). The key contributions of this paper are summarized below.

1. We prove that in wavefront loops, if the accumulation operator is associative and commutative and the distribution operator is either distributive over or same with the accumulation operator, breaking through the data dependency and changing the sequence of computation operators properly does not affect the correctness of results. This provides the guidance for developers under which circumstances, the compensation-based method can be used. (In ③ and ④ of Fig. 1.)
2. We design a highly efficient compensation-based method on GPUs. Our method provides the weighted scan-based GPU kernels to optimize the computation, and combines with the tiling method to optimize the memory access and synchronization. (In ④ of Fig. 1.)
3. We carry out a comprehensive evaluation on both kernel level and application level to demonstrate the efficiency of our method over the state-of-the-art research for wavefront loops.

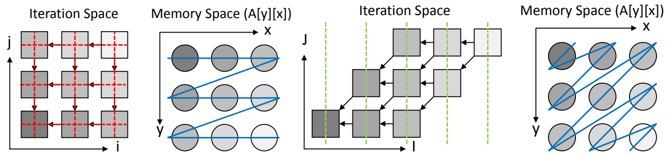
II. Background and Motivation

A. Wavefront Loops and Direct Parallelism

When loop-carried dependencies are present, compilers are oftentimes hard to parallelize the loops effectively, even with the auto-vectorization technologies and user-provided directives [14]. Alg. 1 shows an example of the original loop nests with such data dependencies. The corresponding iterations and memory spaces are shown in Fig. 2a. This loop can be parallelized for neither the *i*-loop nor *j*-loop, if we admit the row-major memory access pattern[1]. Fortunately, the parallelism-inhibiting dependencies can be 'eliminated' by applying loop transformation techniques, e.g., loop skewing and loop interchange [2, 1]. The transformed loop is also shown in Alg. 1. Thereafter, the potential parallelism can be exposed from the iteration space in Fig. 2b. However, this approach has two significant drawbacks: (a) load imbalance, especially at the beginning and the ending iterations (shown in the iteration space); (b) non-contiguous memory access (shown in the memory space).

Algorithm 1 Original & transformed loop nests with wavefront parallelism.

```
// Original
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++)
     A[i][j] = A[i][j-1] * 0.5 + A[i-1][j] * 0.5;
// Transformed via loop skewing and interchange
for(int I = 0; I < m+n-1; I++)
  for(int J = max(0, I-n+1); J < min(m, I+1); J++)
    A[J][I-J] = A[J][I-J-1] * 0.5 + A[J-1][I-J] * 0.5;
```



(a) Original (b) Transformed

Figure 2: Exposed parallelism and corresponding memory access pattern of the two forms of loop nests in Alg. 1. In the iteration space, the arrow represents the data dependency, e.g., a←b means iteration b depends on a.

Alg. 1 also shows there are two types of binary operators in wavefront loops. The *distribution operator* will distribute a part of the value of one entry to another. For example, in this example, the multiplication "$*$" is the distribution operator, which distributes a portion of `A[i][j-1]` and `A[i-1][j]` to `A[i][j]`. Another operator is the *accumulation operator*, which will accumulate incoming values into an entry. Here, the accumulation operator is "+". Most existing studies strictly follow the sequence of operators, which means an entry will be updated by the accumulation operator only after receiving the distributed values from all prerequisites.

B. Tiling-based Solutions and Their Limitations

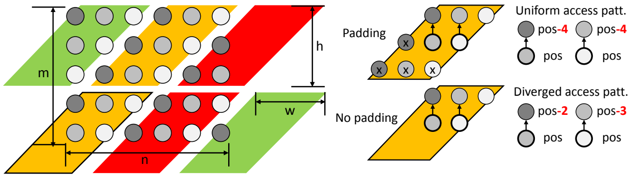
To reduce the cost of load imbalance and amortize the overhead of non-contiguous memory access, many studies [5, 15, 3] apply the tiling-based methods, where the spatial locality can be improved and the expensive synchronization among each entry will convert to the synchronization among tiles. However, there are two other issues.

Data layouts: In a basic design using the tiling optimizations, one can divide the working set into tiles and follow the anti-diagonal direction to parallelize the computation. The overhead of accessing non-contiguous data is mitigated by the cache. However, the non-contiguous data access still

[1] By default, we assume the row-major layout for all arrays.

exists inside each tile, and at the beginning and the ending of anti diagonals, there are no enough entries that can be executed in parallel. This motivates the anti-diagonal major storage and *hyperplane* on GPUs [4, 5]. However, there are two another problems emerging:

(1) Wasted memory and computing resources. Suppose the dimensions of the working matrix $\mathtt{A}$ are m by n and it is divided into *hyperplane* tiles of h by w, shown in Fig. 3a. To store the array $\mathtt{A}$, we need to allocate $\lceil \frac{m}{h} \rceil \cdot \lceil \frac{n+h-1}{w} \rceil$ *hyperplane* tiles, where $n+h-1$ is to process n row entries plus the longest preceding padding entries that is equal to $h-1$. As a result, the actual memory usage for all *hyperplane* tiles must be larger than $m \cdot (n+h)$. Therefore, in the *hyperplane* mode, the percentage of effective memory usage is approximately $n/(n+h)$. Apparently, the padding overhead is not negligible when $n+h$ is sufficiently larger than n.



(a) Hyperplane tiles (b) Access patterns

Figure 3: Splitting the array $\mathtt{A}$ into hyperplane tiles and their access patterns w/ and w/o padding.

One might wonder that the padding could be removed by adding rules to skip out-of-bound access. However, this strategy will break the uniform access that is preferred by GPUs. Fig. 3b demonstrates the diverged access in the padding-free scenario. Compared to the uniform pattern, each highlighted element needs to use different indexing formulas to obtain their north neighbors, e.g., $pos - 2$ and $pos - 3$ in the figure. Therefore, the padding-free strategy may greatly increase the complexity of indexing and lead to more branches in GPU kernels, resulting in the performance degradation.

(2) Layout transformation overhead. To remove the non-contiguous memory access, the data layout can be transformed from the row-major to the anti-diagonal major [16, 5]. However, this conversion not only requires developers refactoring the implementations, but causes significant transformation overhead. In the evaluation, we have observed the transformation time makes up to 31˜60% and 40˜72% of computation time on on NVIDIA K80 and P100 GPUs, respectively (Sec. VI-B2).

Task scheduling: The tile-based solutions, e.g., [3, 5, 17], assigns a complete row of tiles to one compute unit, e.g., a Multiprocessor (MP) of GPU. In that way, the sequential execution order by a MP naturally satisfies the dependency between tiles on the same row; while the dependencies between tiles on different rows can be satisfied via lightweight local synchronizations, e.g., the spin-lock, leading to a pipeline-like execution mode. This methodology works very well for square matrices (e.g., $m \approx n$), because the load balance among compute units can be quickly achieved by the large amounts of parallel tiles along the anti-diagonal. However, for rectangular matrices, especially when $m \ll n$, such a methodology may lose the efficiency, since there are no sufficient tiles in most anti-diagonals.

C. *Compensation-based Solutions and Their Limitations*

In recent years, several studies [7, 18, 6] have offered another type of solutions for parallelizing wavefront loops. Overall, the computation is conducted in a row-by-row manner. The contiguous data entries in a row are divided into groups and scheduled to different compute units. Fig. 4 shows three main steps on processing the bottom row: (1) each compute unit ignores the horizontal data dependency and computes its data entries in parallel to generate the intermediate results; (2) a compensation step is performed to compute ignored data for each entry in a scan-like process; (3) each compute unit corrects the intermediate results with the compensations for the final results. Each step of this method can be parallelism-friendly, and also easy to balance entries between compute units.

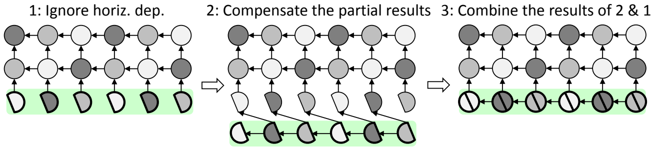

1: Ignore horiz. dep.
2: Compensate the partial results
3: Combine the results of 2 & 1

Figure 4: Compensation-based solutions decompose the processing into three steps, each of which can be parallelism-friendly and load balanced.

However, this solution requires multiple expensive global synchronizations within and between processing each row. Within a row, in the step 2, after a compute unit finish its local computation on the ignored data, it has to wait for the finish of all preceding compute units to get their compensation results, because the data dependency is propagated from the start to the end along a row. Between rows, only after the third step finishes at all compute units, they can continue processing the next row to avoid the data dependency between rows. Thus, the performance might deteriorate without a highly optimized compensation design. More importantly, previous research illustrates the compensation-based parallelism works well for string matching operators, e.g., *max* and +, but its generality to other domains is still unclear. As a consequence, in this paper, we will determine the boundary of the compensation-based method and answer the question: under which circumstances, can the compensation-based method be used to optimize wavefront loops?

III. Compensation-based Computation – Theory

A. *Standard Wavefront Computation Pattern*

We define our target wavefront computation pattern by capturing the key operations and formalizing their data dependencies.

Definition 1. (*Wavefront Pattern*) Let $A = (A_{i,j})$ be a m by n matrix to store the output of the wavefront computation. For any entry $A_{i,j}$, where $0 < i < m$ and $0 < j < n$, the relationships with its neighbors, e.g., $A_{i,j-1}$, are defined by applying two generic binary operators $\diamond$ and $\circ$ as shown in Eq. I. Besides, a constant or variable value b can be applied on the operator $\circ$. Note, when $i = 0$ or $j = 0$, $A_{i,j}$ can be predefined according to application-specific rules.

$$A_{i,j} = (A_{i,j-1} \circ b_0) \diamond (A_{i-1,j} \circ b_1) \diamond (A_{i-1,j-1} \circ b_2) \tag{I}$$

In the definition, $\circ$ is the distribution operator and $\diamond$ is the accumulation operator, while we will use these symbolic representations in the proof. This abstracted definition can cover various real-world wavefront loops by transforming the generic operators into concrete ones. The practical cases will be discussed in Sec. IV.

B. Compensation-based Computation Pattern

We can present the compensation-based computation pattern with the generic operators in Eq. I. First, the data dependencies along the j-direction are ignored and the partial results are represented as $\tilde{A}_{i,j}$, leading to Eq. II-1. Second, an additional compensation step of Eq. II-2 is carried out to produce the compensation values $B_{i,j}$. Third, the compensation values are used to correct the partial results $\tilde{A}_{i,j}$ for the loss caused by the loosened dependencies, as shown in Eq. II-3. The symbols $\prod$ and $\sum$ represent the iterative binary operations $\circ$ and $\diamond$, respectively.

$$\tilde{A}_{i,j} = (A_{i-1,j} \circ b_1) \diamond (A_{i-1,j-1} \circ b_2) \tag{II-1}$$

$$B_{i,j} = \begin{cases} \sum_{u=0}^{j-1}(\tilde{A}_{i,u} \circ \prod_{v=u}^{j-1} b_0) & \text{when } \circ \neq \diamond \\ \sum_{u=0}^{j-1}(\tilde{A}_{i,u} \diamond b_0) & \text{when } \circ = \diamond \end{cases} \tag{II-2}$$

$$A_{i,j} = \tilde{A}_{i,j} \diamond B_{i,j} \tag{II-3}$$

Obviously, this new pattern has changed the computation ordering in Eq. I. Thus, to show the validity, we need to prove that under which circumstances, the Eq. II-3 (with the Eq. II-1 and Eq. II-2) is equivalent to the Eq. I.

Theorem 1. The compensation-based computation shown in Eq. II-3 (incl. Eq. II-1 and II-2) is equivalent with the original computation in Eq. I, provided the binary operators $\diamond$ is associative and commutative, and (1) $\circ$ has the distributive property over $\diamond$, or (2) $\circ$ is same with $\diamond$ (where, for brevity, we only use $\diamond$).

Proof: We use the induction method to prove the equivalence of the two equations. First, we focus on a *base case* to prove the statement holds for updating the first element $A_{1,1}$. Starting from Eq. II-3, we have $A_{1,1} = \tilde{A}_{1,1} \diamond B_{1,1}$. According to Eq. II-2 and $A_{1,0}$ is predefined, the item $B_{1,1} = \tilde{A}_{1,0} \circ b_0 = A_{1,0} \circ b_0$, no matter $\circ$ is same with $\diamond$ or not. Then, putting Eq. II-1 and Eq. II-2 into Eq. II-3, we can get $A_{1,1} = (A_{0,1} \circ b_1) \diamond (A_{0,0} \circ b_2) \diamond (A_{1,0} \circ b_0)$. Since $\diamond$ has the commutative property, this is equal to $(A_{1,0} \circ b_0) \diamond (A_{0,1} \circ b_1) \diamond (A_{0,0} \circ b_2)$, which is $A_{1,1}$ defined by Eq. I. Thus the statement is true for the base case.

Then, we focus on the *inductive step*: if the statement holds for $j = k - 1$, then it also holds for $j = k$.

In the case of $\circ \neq \diamond$, based on the Eq. II-2, we know $B_{i,k} = \sum_{u=0}^{k-1}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-1} b_0)$. We unfold $\sum$ to get:

$$B_{i,k} = \sum_{u=0}^{k-2}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-1} b_0) \diamond (\tilde{A}_{i,k-1} \circ b_0) \tag{1}$$

Since $\diamond$ has the commutative and associative properties, this can be transformed to:

$$B_{i,k} = (\tilde{A}_{i,k-1} \circ b_0) \diamond (\sum_{u=0}^{k-2}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-1} b_0)) \tag{2}$$

Since $\circ$ has the distributive property over $\diamond$, we can "factor out" a b_0 from each term and get:

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond (\sum_{u=0}^{k-2}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-2} b_0))) \circ b_0 \tag{3}$$

Using Eq. II-2 when $j = k - 1$, Eq. 3 can be simplified to:

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond B_{i,k-1}) \circ b_0 \tag{4}$$

Because the induction hypothesis that $j = k - 1$ holds, meaning $A_{i,k-1} = \tilde{A}_{i,k-1} \diamond B_{i,k-1}$ is true, we can get:

$$B_{i,k} = A_{i,k-1} \circ b_0 \tag{5}$$

Then, putting Eq. II-1 and Eq. 5 to Eq. II-3, we get $A_{i,k} = (A_{i-1,k} \circ b_1) \diamond (A_{i-1,k-1} \circ b_2) \diamond (A_{i,k-1} \circ b_0)$. Due to the commutative property of $\diamond$, this is equal to Eq. I. Therefore, we demonstrate the statement also holds for $j = k$ in the case of $\circ \neq \diamond$.

Now, we consider the case of $\circ = \diamond$, where $B_{i,k} = \sum_{u=0}^{k-1}(\tilde{A}_{i,u} \diamond b_0)$. Then, due to the associative and commutative property of $\diamond$, $B_{i,k}$ can be transformed to:

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond (\sum_{u=0}^{k-2}(\tilde{A}_{i,u} \diamond b_0))) \diamond b_0 \tag{6}$$

Eq. 6 can be simplified by using Eq. II-2 when $j = k - 1$.

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond B_{i,k-1}) \diamond b_0 \tag{7}$$

Using the induction hypothesis that $j = k - 1$ holds, we can get $B_{i,k} = A_{i,k-1} \circ b_0$. Then, similar to the case of $\circ \neq \diamond$, we can prove Eq. II-3 is equal to Eq. I for the case of $\circ = \diamond$ in $j = k$.

Since both the base and inductive cases have been performed, the statement holds for all natural numbers j. ■

Now, we compare the complexity of the proposed compensation-based method with the original one. Obviously, the key difference of the two methods relies on how to satisfy the dependencies along the j-direction. In the original method, it is done by $A_{i,j-1} \circ b_0$ with $O(1)$ complexity, while in the proposed method, Eq. II-2 is used for the same purpose, leading to $O(n)$ complexity. Nevertheless, Eq. II-2 can be also optimized to $O(1)$ by using dynamic programming techniques, i.e., $B_{i,j} = (B_{i,j-1} \circ b_0) \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for $\circ \neq \diamond$ or $B_{i,j} = B_{i,j-1} \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for $\circ = \diamond$. However, updating $B_{i,j}$ is still more expensive than the original $A_{i,j-1} \circ b_0$ operation. We will show the proposed method can expose more parallelisms in Sec. V, and thus it provides better performance in Sec. VI.

IV. Compensation-based Computation – Practice

Here, we discuss representative wavefront loops and how these loops can be expressed in the compensation-based parallelism patterns from Sec. III.

SOR Solver (SOR) [8]: The successive over-relaxation (SOR) conducts stencil-like computation to solve a linear system of equations in an iterative fashion. As below, `A[i][j]` represents a discrete gridpoint and its new value depends on its neighbors: some are the most recently updated (i.e., `A[i][j-1], A[i-1][j]`), while others are from the previous time step (i.e., `A[i][j], A[i+1][j], A[i][j+1]`), resulting in a wavefront computation pattern.

```
A[i][j] = (A[i][j] + A[i][j-1] + A[i-1][j] +
           A[i+1][j] + A[i][j+1]) / 5;
```

To express the computation in the compensation-based parallelism pattern in Sec. III, we map ($\diamond$, $\circ$) to $(+, \cdot)$ and b_0, b_1, b_2 to 0.2. Obviously, the operator $+$ and $\cdot$ satisfy the requirements of the Thm. 1.

Smith-Waterman (SW) [9]: It is a well-known algorithm to align the input sequences a and b. `A[i][j]` stores the maximum score for aligning the sub-sequences 0-i of a and 0-j of b. The `s(i,j)` is the substitution function (i.e., b_2) to check if the corresponding amino acids are same at i of a and j of b. The constant `2` is the insertion/deletion penalty (i.e., b_0 and b_1). For the operators, we map ($\diamond$, $\circ$) to $(max, +)$. Note, max is a binary operator, but for brevity, we put four operands in this form.

```
A[i][j] = max(A[i][j-1] - 2, A[i-1][j] - 2,
              A[i-1][j-1] + s(i,j), 0);
```

Summed-area Table (SAT) [10]: It is used to accelerate texture filtering in image processing, where `A[i][j]` stores the sum of all pixels above and to the left of the point (i,j). Thus, `p[i][j]` is the pixel value (i.e., b_2). In addition, the operator $\circ$ is equal to $\diamond$ and is $+$; b_0 and b_1 are both 0. Note that, the computation order in the compensation-based method discussed in the previous section is along the j-direction. In this case, all entries at the row $i-1$ have been updated when processing the row i. As a result, the negation on `A[i-1][j-1]` will not affect the correctness.

```
A[i][j] = p[i][j] + A[i][j-1] + A[i-1][j] - A[i-1][j-1];
```

Integral Histogram (IHist) [11]: It extends the SAT and enables the multi-scale histogram-based search. In this method, `A[i][j]` is the histogram position for a bin z of its top-left sub-image, and thus, `Q(i,j,z)` checks if the pixel (i,j) belongs to bin z or not (i.e., b_2). The other parts are similar with SAT: $\circ$ is equal to $\diamond$ and is $+$; b_0 and b_1 are both 0.

```
A[i][j] = A[i][j-1] + A[i-1][j] - A[i-1][j-1] + Q(i,j,z);
```

V. Design and Implementation on GPUs

This section presents our efficient design of the compensation-based parallelism on GPUs. Because Eq. II-1 and Eq. II-3 naturally present no dependencies between neighboring entries and are easy to parallelize, we focus on Eq. II-2, the compensation step.

A. Compensation-based Computation on GPUs

For the compensation step, based on Eq. II-2, we can transform the computation to a fixed number of operations, i.e., $B_{i,j} = (B_{i,j-1} \circ b_0) \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for the case of $\circ \neq \diamond$, and$B_{i,j} = B_{i,j-1} \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for the case of $\circ = \diamond$. This method is used by previous research [7]. However, it will make every cell in B to depend on all preceding entries, causing strong data dependencies. Besides, the two formulas indicate different strategies to cope with parallelization, setting obstacles for the implementations on GPUs.

Therefore, we propose the efficient **scan-** and **weighted scan-based** methods to process the compensation computation. For the case of $\circ = \diamond$, because all previous $\tilde{A}$ contribute equally to current B, they only need to add a single b_0 on the operator ($\circ$). For example, to calculate $B_{1,3}$, $\tilde{A}_{1,0} \circ b_0$, $\tilde{A}_{1,1} \circ b_0$, and $\tilde{A}_{1,2} \circ b_0$ are used and each of them only needs to add a single b_0 on the operator without the consideration of the index. This actually corresponds to a typical **scan** operation, which has be well understood on GPUs [19]. However, the case $\circ \neq \diamond$ is much complicated, because each previous $\tilde{A}$ has different impacts on current B. For example, to calculate $B_{1,3}$, $\tilde{A}_{1,0} \circ 3b_0$, $\tilde{A}_{1,1} \circ 2b_0$, and $\tilde{A}_{1,2} \circ b_0$ are applied[2]. Thus, the index (or distance) information of each operand has to be considered. We call this as the **weighted scan** pattern and its parallel design is shown in Fig. 5.

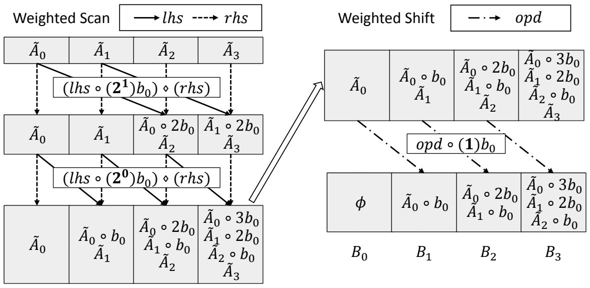

Weighted Scan
$\longrightarrow lhs \dashrightarrow rhs$
$\tilde{A}_0$ $\tilde{A}_1$ $\tilde{A}_2$ $\tilde{A}_3$
$(lhs \circ (\mathbf{2^1})b_0) \diamond (rhs)$
$\tilde{A}_0$ $\tilde{A}_1$ $\tilde{A}_0 \circ 2b_0$ $\tilde{A}_2$ $\tilde{A}_1 \circ 2b_0$ $\tilde{A}_3$
$(lhs \circ (\mathbf{2^0})b_0) \diamond (rhs)$
$\tilde{A}_0$ | $\tilde{A}_0 \circ b_0$ $\tilde{A}_1$ | $\tilde{A}_0 \circ 2b_0$ $\tilde{A}_1 \circ b_0$ $\tilde{A}_2$ | $\tilde{A}_0 \circ 3b_0$ $\tilde{A}_1 \circ 2b_0$ $\tilde{A}_2 \circ b_0$ $\tilde{A}_3$
Weighted Shift
opd
$opd \circ (\mathbf{1})b_0$
ϕ | $\tilde{A}_0 \circ b_0$ | $\tilde{A}_0 \circ 2b_0$ $\tilde{A}_1 \circ b_0$ | $\tilde{A}_0 \circ 3b_0$ $\tilde{A}_1 \circ 2b_0$ $\tilde{A}_2 \circ b_0$
B_0 B_1 B_2 B_3

Figure 5: Parallel design of the weighted scan-based compensation computation. The operands lhs and rhs represent the left-hand side and the right-hand side of the $\diamond$ operator.

In the design, we conduct the compensation computation in two stages. (1) the weighted scan. This is an iterative computation to consider the effects of each preceding operand lhs on the current operand rhs. Suppose the size of the input array $\tilde{A}$ is n, we need $log_2 n$ steps (from $log_2 n - 1$ to 0) to finish the weighted scan. In each step, an entry lhs will contribute the weight $(2^i)b_0$ to the entry rhs with the distance (2^i). For example, as shown in the figure, when the input array is 4, there are 2 steps in the weighted scan. In the step $i = 1$, the lhs operand $\tilde{A}_0$ contributes $(2^1)b_0$ to the rhs operand $\tilde{A}_2$, leading to $(\tilde{A}_0 \circ (2^1)b_0) \diamond \tilde{A}_2$ on the position of $\tilde{A}_2$. Note, if the position of rhs is less than the current

[2] In the following sections, for brevity, the coefficient before b_0 means the number of $\circ$ operations on b_0, e.g., $2b_0$ equals to $\prod_{k=1}^{2} b_0$.

distance 2^i, the lhs doesn't need to contribute anything to rhs. (2) the weighted shift. According to Eq. II-2, $B_{i,j}$ stores only the summation of previous weighted $\tilde{A}_{i,u}$ with u up to $j-1$. Thus we need to compensate the previous results from the weighted scan to eliminate the effects of current operand. This can be achieved by shifting each item and add an additional weight b_0, as shown in the right part of Fig. 5.

In order to better fit the underlying architecture of GPU, our implementation carries out the warp-aware SIMD computation by explicitly operating data at the register level [20, 21]. Alg. 2 shows our weighted scan GPU kernel for the operator $(+, \cdot)$. The function `blk_wscan` in line(L) 2 processes data assigned to each block. First, we load current data to `rhs` (L8-9), followed by a series of warp-level shuffle operations to realize the inner-warp weighted scan (L11-15) using the scaling weight `w` with the distance `i` (L13). The intermediate results are stored on shared memory. Then, a single thread will handle the inter-warp weighted scan over the intermediate results, where the weight grows by the distance of a `WRP_SIZE` (L18-26). Finally, we broadcast the values on shared memory (representing the effects from preceding warps) to the local value in current warp; still, the weight should be scaled up with the local index (L29). Note, in L31, `f` determines if an additional weight `w` is needed for modifying the current value (corresponding to the aforementioned weighted shift).

The function `cpst_based_comput` in L34 is a recursive function to deal with the data exceeding a block size and the basic case is identified in L39. The function `blk_wreduce` is a variant of `blk_wscan` to perform weighted reduction operations over the data for each block, whose intermediate results are stored in `part_d`. Then, we recursively call the weighted scan to process `part_d` using the weight with the distance of `BLK_SIZE` (L45). At last, `blk_wscan` is used to carry out the inner-block weighted prefix sum (L47).

Algorithm 2 Weighted prefix sum for the operator $(+, \cdot)$

```
 1 __global__ // BLK_SIZE: block size; WRP_SIZE: warp size
 2 void blk_wscan(float *in, float *out, int n,
 3      float w, float *partial, bool f) {
 4   __shared__ float smem[BLK_SIZE/WRP_SIZE];
 5   // gid: global idx; tid: thread idx; bid: block idx;
 6   // lid: lane idx; wid: warp idx;
 7   float rhs, lhs;
 8   if(tid == 0) rhs = partial[bid];
 9   else rhs = (gid<n)?in[gid-1]:0;
10   /* Inner-warp weighted prefix-sum */
11   for(int i = (WRP_SIZE>>1); i >= 1; i >>= 1) {
12     lhs = __shfl_up(rhs, i);
13     if(lid >= i) rhs = lhs*__powf(w, i) + rhs;
14   }
15   if(lid == WRP_SIZE-1) smem[wid] = rhs;
16   /* Inter-warp weighted prefix-sum */
17   __syncthreads();
18   if(tid == 0) {
19     float lhs2 = smem[0]*w, rhs2;
20     smem[0] = partial[0];
21     for(int i=1; i<BLK_SIZE/WRP_SIZE; i++) {
22       rhs2 = smem[i];
23       smem[i] = lhs2;
24       lhs2 = lhs2*__powf(w,WRP_SIZE)+rhs2*w;
25   }}
26   __syncthreads();
27   /* Inner-warp broadcast */
28   rhs = rhs +
29         ((!lid)?smem[wid]:smem[wid]*__powf(w, lid));
30   /* Modification */
31   if(f) if(tid != 0) rhs *= w;
32   if(gid < n) out[gid] = rhs;
33 }
34 void cpst_based_comput(float *in, float *out, int n,
35     float base, float w, bool f=true) {
36   dim3 blks(BLK_SIZE, 1, 1);
37   dim3 grds(CEIL_DIV(n, BLK_SIZE), 1, 1);
38   // Device malloc part_d for partial results
39   if(dimGrid.x == 1) {
40     // D2H: copy base to part_d
41     blk_wscan<<<grds, blks>>>(in, out, n, w, part_d, f);
42     return;
43   }
44   blk_wreduce<<<grds, blks>>>(in, out, n, w, part_d, f);
45   cpst_based_comput(part_d, part_d, grds.x,
46     base, pow(w,BLK_SIZE), false);
47   blk_wscan<<<grds, blks>>>(in, out, n, w, part_d, f);
48 }
```

One can also implement the weighted scan-based compensation method on GPUs by leveraging the scan functions from GPU library, e.g., Thrust and ModernGPU. The appendix shows how to prepare corresponding customized comparators for the library-based scan functions, which will be used as one of the baselines in the evaluation.

B. Synchronizations on GPUs: Global vs. P2P

As discussed in Sec. II-C, the compensation-based method may encounter the performance degradation due to the synchronizations within and between rows. The weighted scan method can mostly mitigate the synchronization overhead within a row; while between rows, the synchronization still affects the performance, because existing compensation-based solutions [6, 7] schedule thread blocks in a row-by-row manner, and each thread block have to wait for the finish of all others before processing the next row. We call this scheduling method as the global synchronization. On the contrary, the tile-based solutions [5, 3] use a pipeline-like mode: the tiles in a same row will be assigned to a thread block and be processed sequentially, which guarantees the horizontal dependencies; when the computations on a tile finish, it will trigger the processing on tiles below through a lightweight peer-to-peer (P2P) synchronization, e.g., spin locks, which guarantees the vertical dependencies.

Fig. 6 exhibits these two methods to solve Alg. 1 over different working matrices by varying their dimensions of m and n. When m and n are close to each other, sufficient tile-level parallelism can be exposed, since there are many parallel tiles along the anti-diagonal. Thus, in the tile-based method, the poor performance of sequential processing at the beginning and the ending diagonals can be effectively hidden. In this scenario, i.e., m and n are close, processing the matrix row by row would cause high overhead due to the frequent global synchronizations. This is shown in the right part of Fig. 6. On the other hand, when m is much larger than n, the portion of the beginning and ending diagonals is not negligible in the anti-diagonal-major method; in the extreme case, the whole computation would be serialized. For example, each row/anti-diagonal only contains a single tile, and thus, no tiles can be processed in parallel. In this scenario, as shown in the left part of Fig. 6, the row-major method with the global synchronization can provide much better performance.

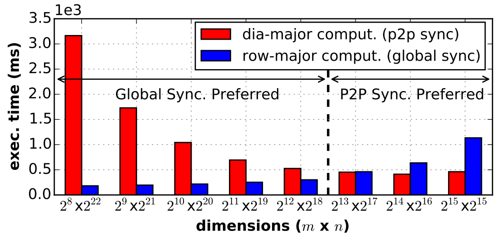


Figure 6: Performance comparison between the row-major computation with global sync. and the anti-diagonal-major computation with peer-to-peer (p2p) sync. The row-major kernel is based on our weight scan based method, while the diagonal-major kernel is based on a tiled solution [5].

C. Putting Them All Together

As discussed in the previous subsection, there is no single scheduling and synchronization method that can fit in different scenarios of workloads. Therefore, we propose a two-level hybrid method for wavefront problems, as shown in Fig. 7. We use the compensation-based computation for the matrices which can expose sufficient parallelism for each row and the row number is limited to reduce the overhead of the global synchronization; otherwise, we switch to a tile and compensation hybrid method that organizes data into tiles and utilize p2p synchronization between tiles, while inside each tile, still uses the compensation-based method to accelerate the computation. To find the optimal switching points, we build a simple offline auto-tuner based on the logistic regression to learn how the software and hardware configuration factors, including the operational intensity on each entry, the m and n of working matrix, and the generation of GPUs, determine the switch point. In the evaluation, we take 200 combinations of factors to determine the likelihood function offline.

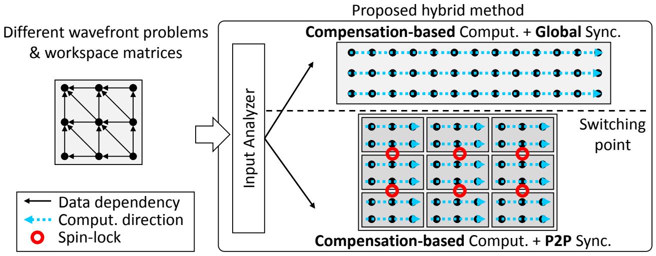


Figure 7: Proposed hybrid method to adapt the computation and synchronization to different wavefront problems and workspace matrices.

VI. Evaluation

We conduct the experiments on two generations of NVIDIA GPUs, i.e., Tesla K80 and P100. The specifications are listed in Tab. I. First, we evaluate the performance of the core kernel in the compensation-based solution. Then, we investigate how the tile sizes affect the performance of our hybrid method. Finally, we report on the performance of the wavefront problems solved by our method compared with state-of-the-art optimizations.

Table I: Experiment Testbeds

	Tesla K80-Kepler	Tesla P100-Pascal
Cores	2496 @ 824 MHz	3584 @ 405 MHz
Multiprocessors (MP)	13	56
Reg/Smem per MP	256/48 KB	256/48 KB
Global memory	12 GB @ 240 GB/s	12 GB @ 720 GB/s
Software	CUDA 7.5	CUDA 8.0

A. Performance of Compensation-based Kernels

We first study the weighted scan performance in the compensation-based solution by comparing the performance of our own design in Alg. 2 with the scan functions based on Thrust and ModernGPU. The customized comparators for the library-based solutions are shown in appendix. As indicated by the four wavefront problems in Sec. IV, we only need three combinations of binary operators, i.e., $(+,\cdot)$, $(max,+)$, and $(+,+)$. The input matrices contain random sizes of m and n varying from 2^{14} to 2^{28}.

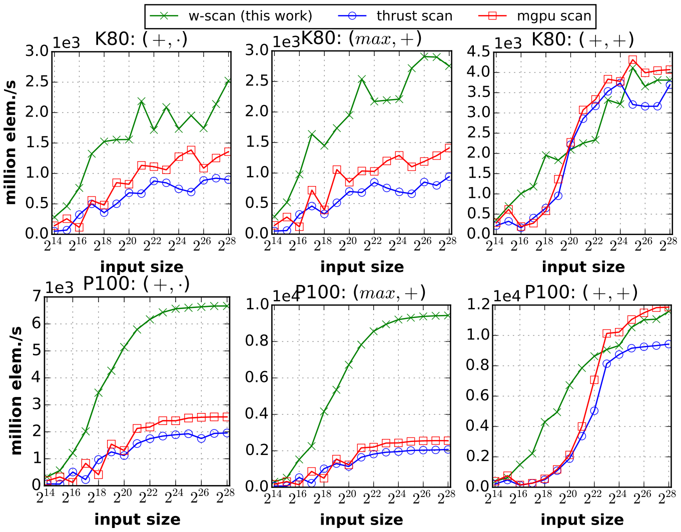


Figure 8: Throughput comparison of the weighted scan kernels.

As shown in Fig. 8, for the case of $\circ \neq \diamond$, i.e., $(+,\cdot)$, $(max,+)$, our design can yield significant performance improvements over Thrust and ModernGPU, achieving an average of 3.5x (4.7x) and 2.4x (3.9x) speedups on K80 (P100). The implementation of ModernGPU explicitly exploits GPU register for data reuse and permutation[3]. However, our implementations not only take advantage of GPU registers, but also optimize the performance due to the following two reasons: (1) we calculate the distance-related weights more efficiently within the kernels; while the library-based methods put all the checking and calculating in the comparators, leading to redundant computations; (2) our algorithm directly operates on the original data and keep track of their

[3] Since ModernGPU is an open source library, our analysis for library-based solutions is mainly based on it.

location in GPU kernels; while the library-based design has to pack and unpack such information before and after the actual computation, causing extra performance penalty.

For the case of $\circ = \diamond$, i.e., $(+, +)$, where there is no need to deal with the varied weights, our solution falls back to a typical scan algorithm and can achieve comparable performance to the highly-optimized library codes. We also observe that our design is particularly effective for the middle range of input sizes. For example, it can deliver an average 5.5x improvements for the inputs ranging from 2^{16} to 2^{22} on P100. This is due to the different parallel strategies. In ModernGPU, each thread is "coarsened" to handle multiple data elements to better utilize the on-chip memory. However, this might result in the degraded GPU occupancy that less threads can be running in a multiprocessors (MP) to hide memory latency for middle-sized inputs. As a contrast, considering the potential heavy use of registers for the weight computation, we schedule a thread to process one element at one time. Besides, this thread-data scheduling strategy can also avoid uncoalesced memory transaction.

B. Performance of Hybrid Kernels

1) Optimal Tile Sizes: Our hybrid method conducts compensation-based computation in tiles when sufficient parallelism is available on anti-diagonals. Thus we first investigate how the tile sizes influence the performance. In the experiment, a large square matrix of 2^{15}x2^{15} is used to represent the case with sufficient anti-diagonal parallelism. In addition, we maximize the shared memory usage (40 KB for each block and other 8 KB for the inter-warp weighted prefix-sum), by using the *persistent thread block* mechanism [5, 17], where each MP only hosts one thread block to avoid deadlock for the spin-lock in the p2p synchronization. The tile sizes (height * width) are shown in Fig. 9. The width corresponds to the thread block size, meaning threads will perform the row-major compensation-based computation. In Fig. 9, we observe that our hybrid method prefers rectangular tiles, because they allow more threads to handle entries in parallel and the resources of registers and shared memory for intermediate values can be more efficiently utilized. For the SOR and SW, the complex weight computation needs more thread warps per block to exploit the high parallelism and data reuse in registers. In contrast, for the SAT and IHist, the computation is relatively simple and small blocks are sufficient. In the following experiments, we set tile sizes to 10x1024 for SOR and SW on K80 and P100, and 20x512 (40x256) for SAT and IHist on K80 (P100). For the other tiling-based solutions, i.e., *tile* and *hypertile* method in Sec. VI-B2, similar tuning procedures are performed, and we select the best tile sizes for them (i.e., the tiles 80x128 or 40x256).

2) Comparison to Previous Work: Now, we evaluate the wavefront problems optimized by our method and state-of-the-art solutions. We fix the total size of the working matrix A to 2^{30} with varying dimensions as shown in Fig. 10. The dashed vertical lines mark the switching points in our hybrid method to use the global and p2p synchronization, whose calculation is based on the auto-tuner presented in Sec. V-C. For the tiling-based methods, the *tile* kernel uses the original row-major data layout, while the *hypertile* uses the hyperplane tiles with the anti-diagonal major layout via the affine transformation. Most of the codes can be found in previous research [5]. For the library-based strategy, *lib-mgpu* and *lib-thrust* are the compensation-based solution with the global synchronization using ModernGPU and Thrust libraries, respectively.

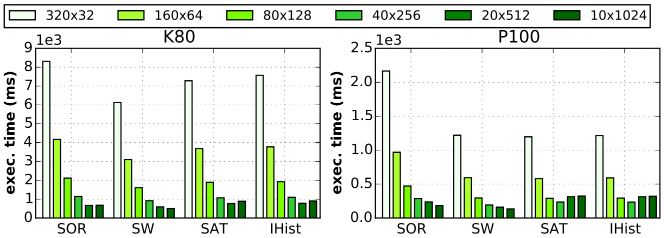


Figure 9: Performance of our hybrid method with varying tile sizes (height * width)

We first focus on the left parts of the vertical dashed lines, where our hybrid methods use the weighted scan with the global synchronization. For SOR and SW, the library-based solutions can achieve an average of 3.7x (4.3x) speedups over tiling-based ones on K80 (P100), because a matrix (height * width) with the longer width can expose more parallelism in a row and at the same time the shorter height places less demands on global synchronization. These scenarios will cause severe serialization of tiling-based solutions, which explains the drastic improvements from our solution of up to 22.1x (43.3x) speedups on K80 (P100). Compared to the library-based solutions, our design can provide an average of 1.9x and 3.1x speedups on K80 and P100, respectively. This can mainly attribute to our native support to the complex weight computation and elimination of pack and unpack overhead, as discussed in Sec. VI-A. For SAT and IHist, our design can deliver the significant speedups: up to 4.8x (6.5x) speedup over the library-based solutions on K80 (P100) when the width of the input matrix falls into the range of 2^{16} to 2^{22}, which is consistent with the results in Sec. VI-A.

Then, we focus on the right parts of the dashed lines, where our hybrid methods switch to the p2p synchronization. In these cases, the performance of library-based solution deteriorates significantly, as more expensive global synchronizations are required. As a contrast, the tiling-based solutions exhibit superior performance as the square-like matrices contain more parallel tiles along anti-diagonals. Compared to the *tile* kernel, our solution takes advantage of both row-major computation and lightweight local synchronization, achieving an average of 3.6x (3.1x) speedups on K80 (P100). For the *hypertile* kernel, its computation is improved significantly for the square-like matrices due to

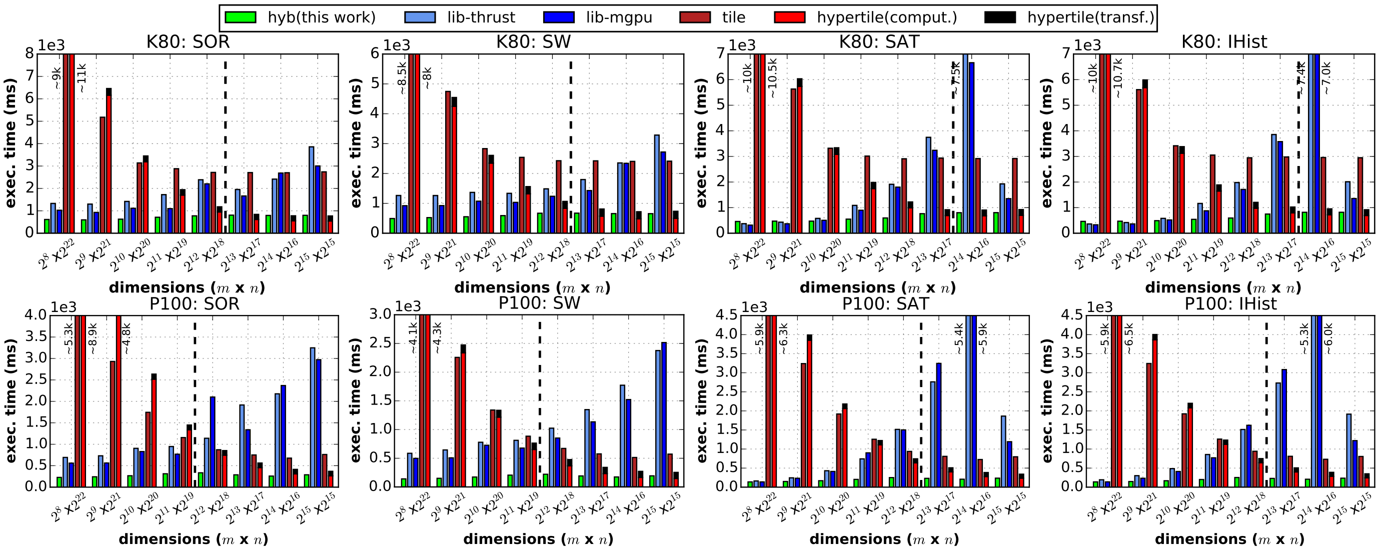


Figure 10: Performance comparison of the library-based (*lib-thrust, lib-mgpu*), tiling-based (*tile, hypertile*) and our hybrid method on different input matrices (height * width). The transformation of data layouts in *hypertile* is also presented. The vertical dashed lines indicate the switch points in our method.

the increased number of entries in each tile that exposes more parallelism opportunities; however, the transformation overhead becomes non-negligible. Our method, by contrast, is able to provide an average of 1.1x speedup on K80 if the transformation overhead is considered, and on P100 we can achieve to an average of 1.8x speedup. Even if we only consider the computation part, our method can still yield an average of 1.3x speedups on P100.

C. Discussion

Precision: For the integer datatype, our compensation-based method can obtain exactly same results with the original methods, e.g., tiling-based ones. However, we also need to consider the precision for float and double datatypes, because changing the computation order may lead to different rounding results. In SOR experiments, we observe the small relative errors are around 10^{-6} if the `float` datatype is used. The error can be further reduced to 10^{-8} for `double` datatype. We believe this is acceptable to the applications using float and double datatypes.

Generality: Thm. 1 poses the requirements on the operators $\circ$ and $\diamond$ along the horizontal dimension; however, in practice, the requirements can be loosened or differentiated along the vertical and diagonal dependencies (e.g., the negation in SAT from Sec. IV). On the other hand, the proof demonstrates this compensation-based method only relies on standard properties of binary operators. Therefore, it could benefit applications in a more general data dependency (e.g., FSM [22]) than the wavefront pattern that only has the dependencies with the horizontal, vertical, and anti-diagonal neighbors.

VII. Related Work

Many efforts have been devoted to the study of wavefront problems for parallelization. One direction is to exploit loop transformation techniques to expose the potential parallelism hidden in target loop nests. These solutions are usually embedded into compilers for automatic parallelism detection and code generation. Wolfe [2] studies the loop skewing techniques to explore the parallelism in such loop nests. Another more general solution to extract the parallelism is to rely on the polyhedral model, which synthesizes affine transformations to adjust the iteration space. Di *et al.* [4] devise a compiler framework using the polyhedral model to maximize intra-tile parallelism for loop nests with dependencies. Baskaran *et al.* [1] present an code transformation system based on polyhedral optimization to generate efficient GPU codes. Other efforts on this model to explore parallelism from loop nests include [23].

Another direction concerns how to map the exposed parallelism efficiently on parallel machines. Xiao *et al.* [17] propose an atomic-based local synchronization to handle dependencies among tiles. PeerWave [5] is a GPU solution for efficient local synchronization between tiles. For the tiling, it utilizes square tiles and hypertiles respectively. Manjikian and Abdelrahman [3] explore the intratile and intertile locality for the large-scale shared-memory multiprocessors. The synchronization problems are also identified in [24, 25]. All the approaches above perform the computation strictly following the original dependency order, which might cause issues on access, locality, and load balance. Differed from them, we focus on using a different computational order for more parallelism (from the problems) and more efficiency (from underlying GPUs).

For domain-specific problems with wavefront patterns, some parallel approaches on computation refactoring are proposed. Farrar [6] and Khajeh-Saeed *et al.* [7] propose methods in sequence alignment to first ignore the dependencies in one direction and then compensate the intermediate results via additional corrections. In practice, the amount of corrections may depend on the characteristics of input sequences [18]. These work, however, requires expert knowledge on the parallel instructions, which are convoluted and idiosyncratic [26, 27, 28], and only ensures feasibility and efficiency for limited operations (e.g., max). Our work, by contrast, addresses a general wavefront problem by outlining its validity and limitation on different combinations of operators using a mathematical proof.

VIII. Conclusion

In this paper, we target on the compensation-based parallelism for wavefront loops on GPUs. We prove that for the compensation-based method, if the accumulation operator is associative and commutative, and the distribution operator is either distributive over or same with the accumulation operator, breaking through the data dependency and changing the sequence of computation operators properly will not affect the correctness of results. We also propose a highly efficient design of the compensation-based parallelism on GPUs. Experiments demonstrate that our work can achieve significant performance improvements for four wavefront problems on various input workloads.

References

[1] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, *Automatic C-to-CUDA Code Generation for Affine Programs*. Springer Berlin Heidelberg, 2010.

[2] M. Wolfe, "Loops Skewing: the Wavefront Method Revisited," *Int. J. Parallel Program.*, 1986.

[3] N. Manjikian and T. S. Abdelrahman, "Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors," *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2001.

[4] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue, "Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs," in *Int. Conf. on Parallel Process. (ICPP)*, 2012.

[5] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu, "PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization," in *ACM Int. Conf. on Supercomput. (ICS)*, 2015.

[6] M. Farrar, "Striped Smith-Waterman Speeds Database Searches Six Times over Other SIMD Implementations," *Bioinformatics*, 2007.

[7] A. Khajeh-Saeed, S. Poole, and J. Blair Perot, "Acceleration of the Smith-Waterman Algorithm Using Single and Multiple Graphics Processors," *J. Comput. Phys.*, 2010.

[8] P. Di and J. Xue, "Model-driven Tile Size Selection for DOACROSS Loops on GPUs," in *Springer Int. Conf. Parallel Process. (Euro-Par)*, 2011.

[9] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *J. Mol. Biol.*, 1981.

[10] H. Nguyen, *GPU Gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[11] F. Porikli, "Integral Histogram: a Fast Way to Extract Histograms in Cartesian Spaces," in *IEEE Comput. Soc. Conf. on Comput. Vision Pattern Recognit. (CVPR)*, 2005.

[12] J. Hoberock and N. Bell, *Thrust: A Parallel Algorithms Library*, 2015, https://thrust.github.io/ v1.8.1.

[13] S. Baxter, *ModernGPU 2.0: A Productivity Library for General-purpose Computing on GPUs*, https://github.com/moderngpu/moderngpu.

[14] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *Parallel Archit. Compil. Tech. (PACT)*, 2011.

[15] J. Wang, X. Xie, and J. Cong, "Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms," in *IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*, 2017.

[16] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *Int. Conf. High Perform. Comput., Networking, Storage Anal. (SC)*, 2011.

[17] S. Xiao, A. M. Aji, and W. c. Feng, "On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit," in *IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2009.

[18] K. Hou, H. Wang, and W.-c. Feng, "AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors," in *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2016.

[19] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization," in *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2013.

[20] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast Segmented Sort on GPUs," in *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[21] K. Hou, H. Wang, and W.-c. Feng, "GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs," in *ACM Conf. Comput. Front. (CF)*, 2017.

[22] P. Jiang and G. Agrawal, "Combining SIMD and Many/Multi-core Parallelism for Finite State Machines with Enumerative Speculation," in *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2017.

[23] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache, "Tiling and Optimizing Time-iterated Computations on Periodic Domains," in *Parallel Archit. Compil. Tech. (PACT)*, 2014.

[24] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves," in *Int. Conf. Euro-Par: Parallel Process.*, 2016.

[25] X. Wang, W. Liu, W. Xue, and L. Wu, "swSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures," in *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2018.

[26] K. Hou, H. Wang, and W.-c. Feng, "A Framework for the Automatic Vectorization of Parallel Sort on x86-based Processors," *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2018.

[27] K. Hou, H. Wang, and W.-c. Feng, "ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors," in *ACM Int. Conf. Supercomput. (ICS)*, 2015.

[28] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel Transposition of Sparse Data Structures," in *ACM Int. Conf. Supercomput. (ICS)*, 2016.

Appendix
Library-based Implementations

It is invalid to use 'address-of' operator (&) for indexing on GPUs, since the data are controlled explicitly in memory hierarchies with different address spaces. Moreover, restricted by the interface of comparators, we can only define the behavior of two given operands rather than the scan itself. Based on these, we implement the custom comparator in Alg. 3. A new data structure `concat_t` is introduced to associate the original value `v`, its index `i`, and the flag `f` to mark if its distance needs modification (weighted shift). Then, the comparator is shown from line(L) 6: `k` is the distance between the two operands to add weights on `lhsv` in L11 or 14. `f` makes sure the weight is modified if the weighted shift occurs (e.g., L11). The branch in L10 guarantees the `lhsv` is always preceding `rhsv`.

Algorithm 3 Custom comparator in library-based solution for Alg. 1, where the operator combination is $(+, \cdot)$ and the weight is 0.5.

```
 1 typedef struct {
 2   float v; int i; bool f = false;
 3 } concat_t;
 4 template<typename T=concat_t> struct bin_opt {
 5   __device__
 6   T operator()(const T &lhs, const T &rhs) const {
 7     int k = abs(rhs.i - lhs.i);
 8     float lhsv, rhsv;
 9     int idx = max(lhs.i, rhs.i);
10     if(lhs.i < rhs.i) {
11       lhsv = lhs.v * __powf(0.5, lhs.f?k:k+1);
12       rhsv = rhs.v * (rhs.f?1:0.5);
13     } else {
14       lhsv = rhs.v * __powf(0.5, rhs.f?k:k+1);
15       rhsv = lhs.v * (lhs.f?1:0.5);
16     }
17     T res(lhsv+rhsv, index, true);
18     return res;
19 }};
```